Table of Contents

1. 前言
2. 第1章 一个简单的例子
3. 第2章 定义函数
4. 第3章 对象类型
5. 第4章 异常
6. 第5章 结构体封装 && 内存管理
7. 第6章 swig && cffi
8. 第7章 RDoc && GEM

玩Ruby也有段时间了，作为半个Rubist最近在学习编写Ruby的C扩展时发现相关的资料非常的少。首先是在Ruby的官方文档中没有相关的介绍，其次是网上相关的介绍也很少。好不容易在《Programming Ruby》中找到了一点资料也是Ruby1.8的，有些内容已经不能用了，不过还是有些参考价值的。

作为一个程序员，没有文档是件很纠结的事。关于这点我觉得Python就比较好。Python的官方文档不仅详细，而且还有例子。

经过了一段时间的折腾，现在大致也有所了解了。于是乎准备把这些内容整理一下，然后就有了这本电子书。当然如果要在Ruby中调用C，现在来说可以有多种方法实现。而直接使用C来编写Ruby扩展的方法显得略有些过时了，但是不管怎样它也是一种比较有效的方法。

同时本文的示例的代码均可从我的Github获取到： https://github.com/wusuopu/ruby-c-extension-sample

注意：在编写本书时最新版的ruby是2.1.0版本。

最后，如果觉得本文对您有用，欢迎捐赠。

我的比特币地址：1N4NES3183t2tU64aGbkotCmRYymJ1UDn6

一个简单的例子

目录结构

首先介绍一下ruby项目的代码目录结构。通常情况下一个ruby扩展项目的目录结构如下：

```
NEWS
Rakefile
README.rdoc
doc/
ext/
```

`COPYING`为版权信息；`NEWS`包含了发行信息；`Rakefile`定义了rake任务；`README.rdoc`包含了用于生成RDoc文档的头部信息；`doc`目录下为该项目的文档；`ext`目录下为扩展程序的源代码以及`extconf.rb`文件。以上仅为参考，实际中还需根据自己的情况新增或者减少一些文件。

MKMF

MKMF是ruby扩展构建系统的一部分，它用于生成编译C程序所需的头文件和Makefile文件。通常该脚本的文件名为：`extconf.rb`，这个有点类似于python中的`setup.py`脚本。以下是一个简单的例子：

```ruby
require 'mkmf'

RbConfig::MAKEFILE_CONFIG['CC'] = ENV['CC'] if ENV['CC']

extension_name = 'example'

unless pkg_config('library')
  raise "library not found"
end

have_func('some_function', 'library/lib.h')
have_type('some_type', 'library/lib.h')

create_header
create_makefile(extension_name)
```

第1行导入了mkmf模块；

第5行定义了该扩展模块的名称；

第7到9行调用pkg-config检查所需的库是否存在；

第11行调用`have_func`方法检查在对应的库中`some_function`方法是否已定义，如果存在则会在生成的`extconf.h`文件中定义一个名为`HAVE_`的宏；

第12行调用`have_type`方法检查在对应的库中`some_type`结构体是否已定义，如果存在则会在生成的`extconf.h`文件中定义一个名为`HAVE_TYPE_`的宏；

第14行创建`extconf.h`头文件；

第15行创建`Makefile`文件。

`extconf.rb`脚本编写完成之后，可以执行如下命令使用：

```
$ cd ext
$ ruby extconf.rb
$ make
```

Rakefile

extconf 脚本创建完了，接下来是创建 `Rakefile` 。顾名思义Rakefile就是ruby的Makefile，即就是用ruby来编写Makefile的功能。以下是一个简单的例子：

```
require 'rake/clean'

EXT_CONF = 'ext/extconf.rb'
MAKEFILE = 'ext/Makefile'
MODULE = 'ext/example.so'
SRC = Dir.glob('ext/*.c')
SRC << MAKEFILE

CLEAN.include [ 'ext/*.o', 'ext/depend', MODULE ]
CLOBBER.include [ 'config.save', 'ext/mkmf.log', 'ext/extconf.h', MAKEFILE ]

file MAKEFILE => EXT_CONF do |t|
  Dir::chdir(File::dirname(EXT_CONF)) do
    unless sh "ruby #{File::basename(EXT_CONF)}"
      $stderr.puts "Failed to run extconf"
      break
    end
  end
end
file MODULE => SRC do |t|
  Dir::chdir(File::dirname(EXT_CONF)) do
    unless sh "make"
      $stderr.puts "make failed"
      break
    end
  end
end
desc "Build the native library"
task :build => MODULE
```

第9行和第10行分别设置了要删除的文件的列表。 `CLEAN` 变量中定义的文件列表会在 `rake clean` 命令中被删除； `CLOBBER` 变量中定义的文件列表会在 `rake clobber` 命令中被删除。

从第12行到29行定义了 `build` 任务，用于生成扩展模块。

程序示例

接下来通过一个简单的例子来介绍C扩展的编写。创建一个新文件 `my_test.c` 内容如下：

```
#include "ruby.h"

static VALUE mTest;
static VALUE cTest;

static VALUE t_init(VALUE self)
{
  printf("\nCreate a MyTest instance.\n");

  return self;
}

void Init_my_test()
{
  mTest = rb_define_module("MyTest");
  cTest = rb_define_class_under(mTest, "MyTest", rb_cObject);
  rb_define_method(cTest, "initialize", t_init, 0);
}
```

第1行将ruby的头文件导入进来，以便能使用ruby的api。

在该程序代码中定义一个特殊的函数 `Init_my_test`，它是该模块的初始化函数，在模块首次加载时执行。每个C扩展都需要定义一个名为 `Init_<name>` 的函数，这与Python的C扩展相似。

在 `Init_my_test` 函数内部使用 `rb_define_module` 函数定义了一个名为 `MyTest` 的Module，然后再用 `rb_define_class_under` 函数为该Module定义了一个类 `MyTest`。并且对应的初始化函数为 `t_init`。

- rb_define_module： 定义一个Module；
- rb_define_class_under： 在一个Module内定义一个Class；
- rb_define_method： 定义一个实例方法，所需的参数依次为类对象、方法名、对应的C函数以及参数个数；

注意：所有被Ruby调用的C方法都必须返回一个VALUE类型的变量。

以上这段代码等效于如下ruby代码：

```
module MyTest
  class MyTest
    def initialize
      puts "\nCreate a MyTest instance."
    end
  end
end
```

程序写完之后运行命令： `rake build` 进行编译。

最后再通过一个小程序来进行测试：

```
# app.rb
require "./my_test.so"
require "test/unit"

class TestTest < Test::Unit::TestCase
  def test_test
    t = MyTest::MyTest.new
    assert_equal(Object, MyTest::MyTest.superclass)
    assert_equal(MyTest::MyTest, t.class)
  end
end
```

定义函数

在上一节的例子中简单提到了使用`rb_define_method`来定义一个函数。在这一节来进行详细介绍。

首先，函数有两种类型：

- 参数个数固定；
- 参数个数可变。

然后，以下是`rb_define_method`的基本用法：

```
rb_define_method(ClassObj, "name", c_function, num);
```

执行的结果是为ClassObj这个类定义一个名为`name`的实例方法，对应的C函数为c_function，接收num个参数。

- 若num的值为正数，则表示需要传入num个参数；
- 若num的值为负数，则表示该方法的参数个数是可变的；
  - num为-1,则传入的参数是一个C的数组；
  - num为-2,则传入的参数是一个Ruby的数组。

固定参数

对于参数个数固定的情况，它对应的`c_function`基本形式如下：

```
VALUE c_function(VALUE self, VALUE arg1, VALUE arg2)
{
    // ...
}
```

其中`self`为该方法的调用者，其余的均是该方法的参数。

可变参数

可变参数又分为两种情况：传入的是C数组和传入的是Ruby数组。

C数组

对于第一种情况，它对应的`c_function`基本形式如下：

```
VALUE c_function(int argc, VALUE *argv, VALUE self)
{
    // ...
}
```

这种情况比较复杂。

- `argc` 为传入参数的个数；
- `argv` 为传入参数的集合；
- `self` 为方法调用者。

对于这种情况不要直接对`argv`进行操作，而是使用`rb_scan_args`方法：

```
rb_scan_args(argc, argv, format, ...);
```

`rb_scan_args` 是通过 `format` 字符串指定的格式对 `argv` 进行解析，当参数不匹配时会抛出异常。

下面通过与之等效的ruby代码来介绍一下 `format` 的用法：

```
def foo( arg = nil )
rb_scan_args(argc, argv, "01", &arg);
```

这是定义了一个方法，有0个必须参数和1个可选参数；

```
def foo( &block )
rb_scan_args(argc, argv, "0&", &block);
```

这是定义了一个方法，需要一个block类型的参数；

```
def foo( *args )
rb_scan_args(argc, argv, "0*", &args);
```

这是定义了一个方法，可传入任意个数的参数；

```
def foo( arg, opt = nil, *args, &block )
rb_scan_args(argc, argv, "11*&", &arg, &opt, &args, &block);
```

最后这是对上面的整合。

Ruby数组

这种情况比较简单，它对应的 `c_function` 形式如下：

```
VALUE c_function(VALUE self, VALUE args)
{
    // ...
}
```

这里所有的参数都放在ruby类型的数组 `args` 中。

对象类型

在Ruby中一切都是对象，在C语言中要访问Ruby的对象是通过VALUE类型的变量进行引用的。

Ruby中定义了一些内建的类型，由于文档中没有介绍，下面的这些内容都是我直接从源代码的 `ruby.h` 头文件中出来的。列表如下：

- T_NONE
- T_NIL
- T_OBJECT
- T_CLASS
- T_ICLASS
- T_MODULE
- T_FLOAT
- T_STRING
- T_REGEXP
- T_ARRAY
- T_HASH
- T_STRUCT
- T_BIGNUM
- T_FILE
- T_FIXNUM
- T_TRUE
- T_FALSE
- T_DATA
- T_MATCH
- T_SYMBOL
- T_RATIONAL
- T_COMPLEX
- T_UNDEF
- T_NODE
- T_ZOMBIE
- T_MASK

同时C的api也提供了一些方法用于检测对象的类型：

- int TYPE(obj) 返回obj对象所属的内建类型；
- void Check_Type(VALUE obj, int type) 检查对象obj是否属于type类型，如果不属于则会抛出异常；
- char* rb_obj_classname(obj) 返回对象obj所属的类的名字；

接下来再介绍一下几种常用数据类型及其相关的api。

Numbers

在Ruby中有两种数字类型：Fixnum和Bignum。 与数字相关的一些api有：

接口	描述
INT2FIX(i)	将int类型转化为Fixnum对象
INT2NUM(i)	将int类型转化为Fixnum对象或者Bignum对象
LONG2FIX(i)	与INT2FIX相似
LONG2NUM(i)	与INT2NUM相似
FIX2INT(o)	将Fixnum对象转化为int类型
FIX2LONG(o)	将Fixnum对象转化为long类型

NUM2INT(o)	将Fixnum对象或者Bignum对象转化为int类型
NUM2LONG(o)	将Fixnum对象或者Bignum对象转化为long类型

Strings

与C字符串和Ruby字符串相关的一些api有：

接口	描述
VALUE rb_str_new(const char *ptr, long len)	将ptr指针指向的长度为len的字符串转化为Ruby的字符串对象
VALUE rb_str_new_cstr(const char *ptr)	将ptr指针指向的字符串转化为Ruby的字符串对象
char *rb_string_value_cstr(volatile VALUE*)	将Ruby的字符串对象转化为C的字符串
char *StringValueCStr(VALUE)	rb_string_value_cstr对应的宏

Arrays

与数组相关的一些api：

接口	描述
VALUE rb_ary_new(void)	创建新数组
VALUE rb_ary_push(VALUE ary, VALUE item)	在数组结尾插入新的值
void rb_ary_store(VALUE ary, long idx, VALUE val)	在idx的位置插入新的值

Hashes

与散列表相关的一些api：

接口	描述
VALUE rb_hash_new()	新建一个散列表
VALUE rb_hash_aset(VALUE hash, VALUE key, VALUE val)	设置键值
VALUE rb_hash_aref(VALUE hash, VALUE key)	获取一个键的值
void rb_hash_foreach(VALUE hash, int (*callback)(ANYARGS), VALUE farg)	遍历散列表，callback格式为 (*callback)(VALUE key, VALUE val, VALUE in)。同时它的返回值为 ST_CONTINUE 则表示正常操作；为 ST_STOP 则停止遍历；为 ST_DELETE 则删除当前的键值；为 ST_CHECK 则检查在此操作过程中该散列表是否已被修改，如果被修改则停止遍历。

Blocks && Callbacks

代码块：

接口	描述
int rb_block_given_p(void)	如果传递了代码块则返回1, 否则返回0
VALUE rb_yield(VALUE);	Ruby中的yield功能

VALUE rb_yield_values(int n, ...);	同上
VALUE rb_yield_values2(int n, const VALUE *argv);	同上

Proc(lambda)：

接口	描述
VALUE rb_proc_new(VALUE (*func)(ANYARGS), VALUE val)	创建Proc对象
VALUE rb_proc_call(VALUE self, VALUE args)	调用Proc对象

以上是列出了一些常用的api接口，这些内容在Ruby的文档中也没有介绍，基本上都是在 `ruby.h` 文件和 `intern.h` 文件中找到的。这里再发下牢骚，感觉还是Python的文档详细啊。

关于这些api的例子可以参考本章的示例。

异常

当程序执行出错时通常的做法是抛出一个异常，这个异常既可以是内建的异常类型也可以是自定义的异常类型。

内建异常类

内建的异常类型如下：

- rb_eException;
- rb_eStandardError;
- rb_eSystemExit;
- rb_eInterrupt;
- rb_eSignal;
- rb_eFatal;
- rb_eArgError;
- rb_eEOFError;
- rb_eIndexError;
- rb_eStopIteration;
- rb_eKeyError;
- rb_eRangeError;
- rb_eIOError;
- rb_eRuntimeError;
- rb_eSecurityError;
- rb_eSystemCallError;
- rb_eThreadError;
- rb_eTypeError;
- rb_eZeroDivError;
- rb_eNotImpError;
- rb_eNoMemError;
- rb_eNoMethodError;
- rb_eFloatDomainError;
- rb_eLocalJumpError;
- rb_eSysStackError;
- rb_eRegexpError;
- rb_eEncodingError;
- rb_eEncCompatError;
- rb_eScriptError;
- rb_eNameError;
- rb_eSyntaxError;
- rb_eLoadError;
- rb_eMathDomainError;

抛出异常

通常自定义异常类型是 `rb_eException` 或者 `rb_eStandardError` 的子类。先使用 `rb_define_class` 定义一个异常类型，然后再抛出一个异常。

抛出异常的方法：

- void rb_raise(error_class, error_string, ...) 这是比较常用的一个方法；
- void rb_sys_fail(error_string) 根据errno抛出一个异常；

其中 `rb_raise(error_class, error_string, ...)` 的作用与 ruby 中的 `raise Error, string` 相同。

异常处理

在 Ruby 中可以使用 `rescue` 捕获一个异常，在C代码中与之对应的函数为：

- rb_rescue(cb, cb_args, rescue_cb, rescue_args)
- rb_ensure(cb, cb_args, ensure_cb, ensure_args)

`rb_rescue` 方法对应 ruby 的 `rescue`， 它的四个参数依次是， `cb` ： 要执行的操作，格式声明为 `VALUE cb(VALUE args)` ； `cb_args` ： 传递给 `cb` 函数的参数； `rescue_cb` ： 出现异常时处理异常的回调函数，格式声明为 `VALUE rescue_cb(VALUE args, VALUE err)` ； `rescue_args` ： 传递给 `rescue_cb` 函数的参数。

转化为 ruby 的语法如下：

```
begin
  cb(cb_args)
rescue
  rescue_cb(rescue_args)
end
```

`rb_ensure` 方法对应 ruby 的 `ensure`， 与 `rb_rescue` 类似。 `ensure_cb` 格式声明为 `VALUE ensure_cb(VALUE args)`

转化为 ruby 的语法如下：

```
begin
  cb(cb_args)
ensure
  ensure_cb(ensure_args)
end
```

Throw/Catch

- rb_catch(const char*, VALUE(*)(ANYARGS), VALUE)
- rb_throw(const char*, VALUE)

`rb_catch` 方法的三个参数分别为： 要catch的代码块名称； catch 的回调函数； 回调参数。 `rb_throw` 方法的两个参数分别为： 返回的catch代码块名称； 返回值。

关于异常的用法可以参考本章的示例。

结构体封装 && 内存管理

前面介绍过在C扩展中使用 `rb_define_class_under` 定义自定义类。而且如果要定义一些变量的话也可以使用 `rb_iv_set` 定义实例变量，以及使用 `rb_define_variable` 或者 `rb_global_variable` 定义全局变量。通过这种方法可以实现 Ruby 与 C 共享数据。本章介绍另一种共享数据的方法——封装结构体。

结构体封装

在 Python 的 C 扩展中，定义一个自定义类就是通过定义一个结构体来模拟实现的。

在 Ruby 中定义了几个有用的宏可以方便的对结构体进行封装。

```
VALUE Data_Wrap_Struct(VALUE class, void (*mark)(), void (*free)(), void *ptr);
VALUE Data_Make_Struct(VALUE class, c-type, void (*mark)(), void (*free)(), c-type *ptr);
Data_Get_Struct(VALUE obj,c-type,c-type *);
```

`Data_Wrap_Struct` 是将 C 的数据类型 `ptr` 进行封装，并返回一个 Ruby 类型的对象。该对象是对应的 C 类型为 `T_DATA` ，对应的 Ruby 类型为 `class` 。

`Data_Make_Struct` 首先分配内存空间，然后执行 `Data_Wrap_Struct` 操作。

`Data_Get_Struct` 获取原始数据的指针。

内存分配

如果需要在扩展程序中申请内存空间来存储内容，可以直接使用 C 的原生方法： `malloc` 、 `realloc` 、 `calloc` 。不过这样的话就需要记得手动释放申请的内存，以免出现内存泄漏。为了方便起见还是推荐使用 Ruby 定义的几个api:

- ALLOC(type) 分配type类型大小的空间，并返回type类型的指针
- ALLOC_N(type, num) 分配num个type类型大小的空间，并返回type类型的指针
- REALLOC_N(var, type, num) 将var指向的空间重新分配为num个type类型大小的空间，并返回type类型的指针

这几个api与原生的C方法功能类似，只是内存的申请和释放都是由Ruby进行管理，而不是操作系统系统。

注意：使用以上方法申请的内存得使用 `xfree` 进行释放。

示例程序

接下来通过一个例子来进行讲解。首先声明一个结构体类型：

```
typedef struct {
    int i;
} cMyStruct;
```

然后再定义一个类：

```
cTest = rb_define_class("MyStruct", rb_cObject);
```

接着在类的 `new` 方法内部执行操作，将该结构体进行封装：

```
cMyStruct *ptr = ALLOC(cMyStruct);
VALUE tdata = Data_Wrap_Struct(class, 0, t_free, ptr);
```

现在可以在其他地方访问该结构体：

```
cMyStruct *ptr;
Data_Get_Struct(self, cMyStruct, ptr);
```

完成的例子代码可以从 https://github.com/wusuopu/ruby-c-extension-sample 获取到。

SWIG && FFI

在开头有说过直接使用C来编写Ruby扩展的方法可能显得略有些过时了，因为还有其他更加方便的方法可以让Ruby调用C的库。本章就简单的介绍一下另外两个工具——swig和ffi。

SWIG

SWIG是一个开发工具，能够将C、C++与多种语言进行连接。如： PHP、Python、Perl、Ruby等。

使用SWIG的好处是只需要写一份代码就可以实现在多种语言中调用；而缺点是需要学习SWIG自己的编程语法。

以下是摘自官网的一个例子。

首先下载安装SWIG: http://www.swig.org/download.html

然后新建文件 `example.c` ：

```
/* File : example.c */
#include <time.h>
double My_variable = 3.0;

int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}

int my_mod(int x, int y) {
    return (x%y);
}

char *get_time()
{
    time_t ltime;
    time(&ltime);
    return ctime(&ltime);
}
```

`example.i` ：

```
/* example.i */
%module example
%{
/* Put header files here or function declarations like below */
extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
%}

extern double My_variable;
extern int fact(int n);
extern int my_mod(int x, int y);
extern char *get_time();
```

程序编写完成之后，如果是要编译成 Python 扩展则执行命令：

```
swig -python example.i
gcc -c example.c example_wrap.c -I/usr/include/python2.7
ld -shared example.o example_wrap.o -o _example.so
```

如果要编译在 Ruby 扩展则执行命令：

```
swig -ruby example.i
gcc -c example.c example_wrap.c -I/usr/include/ruby-2.1.0/ -I/usr/include/ruby-2.1.0/i686-linux
ld -shared example.o example_wrap.o -o example.so
```

最后通过一个程序来测试一下结果：

```
require './example.so'

puts Example.fact(5)
puts Example.my_mod(7, 3)
puts Example.get_time()
```

FFI

FFI是一个可以直接加载动态链接库的工具。使用 Ruby-FFI 就可以很方便的调用库的方法。

同样的，以下也是摘自官方的例子。首先安装 ffi：

```
[sudo] gem install ffi
```

或者下载源代码进行安装： https://github.com/ffi/ffi

然后运行一个测试程序：

```
require 'ffi'

module MyLib
  extend FFI::Library
  ffi_lib 'c'
  attach_function :puts, [ :string ], :int
end

MyLib.puts 'Hello, World using libc!'
```

这个例子是在 Ruby 中直接调用 libc 库的 `puts` 方法。结果应该是可以看到输出字符串： "Hello, World using libc!"

RDoc && GEM

前面几章介绍了 ruby 扩展的开发方法，最后再来介绍两个有用的工具——RDoc和GEM。

RDoc

RDoc是 ruby 的一款文档生成工具，它可以通过代码中特殊格式的注释来生成文档。

RDoc是随着 ruby 安装的，因此不需要再进行额外安装。

接下来准备使用第2章的例子，为其添加文档注释。

首先在 `rb_define_module` 语句之前添加模块注释：

```
/*
   Document-module:  MyTest

   C扩展的 MyTest 模块.
*/
mTest = rb_define_module("MyTest");
```

然后在 `rb_define_class_under` 语句前添加类注释：

```
/*
   Document-class:  MyTest

   MyTest 模块下的 MyTest 类.
*/
cTest = rb_define_class_under(mTest, "MyTest", rb_cObject);
```

接着就可以使用 `rdoc` 命令来生成文档了，生成 HTML 格式的文档：

```
$ rdoc --main README.rdoc -o doc/site/api README.rdoc ext/my_test.c
```

生成 ri 格式的文档：

```
$ rdoc --main README.rdoc -o doc/ri -f ri README.rdoc ext/my_test.c
```

为了方便起见，可以为 RDoc 添加一个 `rake` 任务。在 `Rakefile` 中添加如下内容：

```
require 'rdoc/task'

RDOC_FILES = FileList["README.rdoc", "ext/my_test.c"]

Rake::RDocTask.new do |rd|
   rd.main = "README.rdoc"
   rd.rdoc_dir = "doc/site/api"
   rd.rdoc_files.include(RDOC_FILES)
end

Rake::RDocTask.new(:ri) do |rd|
   rd.main = "README.rdoc"
   rd.rdoc_dir = "doc/ri"
   rd.generator = "ri"
   rd.rdoc_files.include(RDOC_FILES)
end
```

现在就可以分别使用 `rake rdoc` 命令和 `rake ri` 命令来生成 HTML 格式和 ri 格式的文档了；然后分别使用 `rake clobber_rdoc`

命令和 `rake clobber_ri` 命令来删除已生成的文档文件。

以上只是一个简单的介绍，关于在C扩展中使用RDoc，可以参考文档： http://docs.seattlerb.org/rdoc/RDoc/Parser/C.html

GEM

GEM 是 ruby 的包管理工具，类似于 python 的 pip。我们扩展程序开发完成之后可以通过 GEM 打包并与其他人分享。

如果你的系统中没有安装 GEM 的话，可以通过这个地址下载安装： http://rubygems.org/pages/download

首先在项目根目录下创建一个新文件 `.gemspec` ：

```
SPEC = Gem::Specification.new do |s|
   s.name = "example"
   s.version = "1.0"
   s.date = '2014-08-21'
   s.summary = "C bindings"
   s.description = "C Bindings"
   s.authors = ["Long Changjin"]
   s.email = ["admin@longchangjin.cn"]
   s.files = ["Rakefile", "COPYING", "NEWS", "README.rdoc", "ext/my_test.c", "ext/app.rb", "ext/extconf.rb"]
   s.homepage = "http://www.xefan.com/"
   s.required_ruby_version = '>= 2.1.0'
   s.extensions = "ext/extconf.rb"
end
```

然后执行命令 `gem build .gemspec` 生成 gem 包。在当前目录下应该会生成一个名为 `example-1.0.gem` 的文件。如果想到与他人分享该 gem 包，可以执行命令 `gem push example-1.0.gem` 将该文件上传。